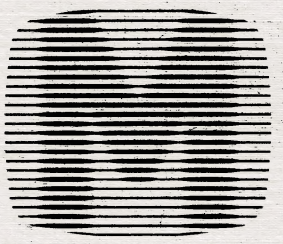


MICROWARE®

OS-9

OPERATING SYSTEM

USER'S GUIDE

Microware Systems Corporation
5835 Grand Avenue
Box 4865
Des Moines, Iowa 50304
515-279-8844

OS-9

OPERATING SYSTEM

USER'S GUIDE

MICROWARE SYSTEMS CORPORATION
5835 Grand Avenue
Des Moines, Iowa 50312
(515) 279-8844

Revision C

Index

INTRODUCTION TO THE OS-9 USERS GUIDE

Welcome to the world of OS-9! If you are new to microcomputers „you will„ be learning in an almost ideal environment. If you are an "old hand", you are about to be introduced to the most modern microcomputer software development technology. In either case, we hope you will have many hours of happy computing.

OS-9 is one of the results of a three-year project to develop a family of software to fully exploit the extraordinary performance of the Motorola 6809 Advanced 8-bit Microprocessor. The design goal of the OS-9 project was to produce a state-of-the-art, extensible, real-time operating system, with particular emphasis on support for modular ROMed software. These goals were not only met, but several important OS-9 features, especially those relating to "memory modules", represent a significant step forward in microcomputer software technology.

The Motorola 6809 microprocessor and OS-9 is the first coordinated hardware/software system that can take advantage of the versatility and cost-effectiveness of the "software on silicon" concept demanded by the complex microcomputer applications of the eighties.

We hope that these new tools can play an important part in makng your job easier.

OS-9's "Parent Processes":

Larry Crane
Bob Doggett
Larry French
Ken Kaplan
Doug Nicholson
Bill Phelps
Terry Ritter

OVERVIEW OF THE USERS GUIDE

OS-9 is a versatile multiprogramming operating system designed to exploit the full capability of the Motorola M6809 Advanced 8-bit microprocessor. When used with the OS-9 family of software, the 6809 can deliver performance equivalent to a mid-size minicomputer.

OS-9 can be used in many ways and forms, from a small "kernel" in ROM used to control a machine tool, up to a fully-equipped timesharing system used in businesses, laboratories and schools. This versatility is one of OS-9's major advantages and distinguishes it from many other operating systems. In particular, OS-9 software is "portable" upward or downward, so you can develop small system software on a big system, or vica-versa.

This guide is designed for persons who will be using an OS-9 system for general interactive applications on a single-user or timesharing system. It is both a tutorial and a reference book. Persons who will be using OS-9 for different applications or who will write assembly-language programs should refer to the "OS-9 System Manual" for detailed information on the inner workings.

OS-9 is available in two versions: Level One is for smaller systems that have up to 65K of memory, Level Two is for systems having up to a megabyte of memory. From the user's point of view, both levels behave almost identically and the information in this guide applies to either version except where otherwise noted.

OS-9 HARDWARE REQUIREMENTS

OS-9 is a modular operating system that can be "configured" in an almost infinite number of ways, therefore "minimum" hardware requirements are hard to define. A small system used for simple controller applications might consist of A 6809 CPU, 4K of ROM and 2K of RAM. A large multi-user Level Two system might have several hard disks, multiple terminals, and 256K bytes of RAM using memory management hardware.

Below are requirements for a "typical" single-user system used for general-purpose software development and similar utility applications.

* 6809 MPU

* 16K RAM memory for assembly-language software
  32K RAM memory for high-level languages
  32K RAM plus a mimimum of 4K per user for timesharing

* 4K bytes of ROM: 2K must be addressed at $F800 - $FFFF, the other 2K is position-independent and self-locating. (some disk versions require 6K rom).

* Disk or cassette tape I/O device

* Console Terminal and interface using serial, parallel, or memory-mapped video display and keyboard

* Optional printer using serial or parallel interface.

* Real time clock hardware

I/O device controller addresses can be located anywhere in the memory space, however it is good practice to place them as high as possible to maximize RAM expansion capability. Standard Microware-supplied OS-9 packages for computers made by popular manufacturers usually conform to the system's customary memory map.

HOW TO INSTALL OS-9: DISK BASED SYSTEMS

Because there are versions of OS-9 customized for many kinds of 6809 computers, this section describes a general installation procedure. Specific instructions for your version of OS-9 is supplied with the software package.

Hardware Preparation

OS-9 systems don't use a "monitor ROM": the OS-9 ROM set is used in its place. If your CPU board came equipped with a monitor ROM, it must be removed and replaced by the OS-9 ROM set.

The first step is to install the OS-9 ROM set, which consists of a total of 4K bytes organized into two sections called "Part 1" and "Part 2". Each part can be either a single 2716-type ROM or a pair of 2708-type ROMS. The "Part 1" ROM(s) must be installed at memory addresses F800 through FFFF, usually on CPU board socket(s). The "Part 2" ROM(s) can be installed at any address, usually also in CPU card socket(s). Some systems have a third disk driver ROM which also goes on the CPU board.

The console terminal, disk controller and other I/O devices must be installed according to the instructions that accompany the software package. Make sure that the interrupt jumpers or switches are enabled on all I/O interfaces.

System Startup

To start the system up, simply turn the computer on: the master disk drive select light should turn on almost at once. Insert the OS-9 system disk in the drive and close the door. OS-9 will load a number of modules into RAM memory, then display a startup message on the system console terminal within several seconds. If this does not happen, reset the computer. If the system does not come up within 30 seconds, turn the computer off and re-check the hardware setup.

WARNING: Never turn power on or off while disks are loaded in drives.

After the shell prints its prompt, use the "setime" utility (if your system uses a real-time clock) to start the clock running. OS-9 cannot perform multitasking (timeslicing) until you have done this.

If the system is being installed for the first time, you may want to create the system password file (see "login" command). If the system is to be used for timesharing, you should also create a shell procedure file for system start-up which runs the "tsmon" utility for each terminal to be used. You can also add other "housekeeping" functions to this file to load special programs, run "tmode" to set terminal parameters, etc.

GETTING STARTED

How you start "talking" to OS-9 depends on which terminal you are using. If you are using the "system console", or if your system has only one terminal, you will be communicating with a program called the "shell". It is called the "shell" because a shell is what surrounds the kernel of a seed: "kernel" is the name used to describe the core of the operating system. These names are descripive of the multi-layered nature of OS-9.

If you are using a terminal other than the system console you may have to "log-on" which involves entering a correct user number and password before you can talk to the shell. An example of the log-on messages and typical responses are shown below. When you log on, you must use the user name and password that was assigned to you. An example is shown below. In it, and elsewhere in this document, the "user's" input is underlined.

```
OS-9 Level 1 Timesharing System  Release 1.Ø   1Ø/Ø5/8Ø 12:33:58

User name?: Bob
Password: open sesame

User 3 logged  1Ø/Ø5/8Ø 12:34:2Ø

Shell

OS9:
```

If you don't enter a valid user name or password, the message "Who" will be displayed and you will be asked to reenter them. You have three chances to enter them correctly. You can shorten the login procedure by typing your user name and password on the same line.

OS-9 is a very complex operating system that has many powerful capabilities. The shell exists to provide an interface between you and the kernel which is easy and convenient to use, without limiting your ability to fully utilize the computer's capabilities. On the other hand, "machine-language" programs communicate directly with the kernel using "service requests" which are 68Ø9 machine-language instructions. The shell's (and command programs') function is to provide most of the same services, using command lines of English-like words that are easier for people to use and understand.

The shell operates by accepting a command line entered from your keyboard and executing it. You will know when the shell is waiting for input because it always displays the prompt "OS9: ". You respond by typing an command line and carriage return after the prompt. For example, the command to run BASICØ9 is this:

```
OS-9: BASICØ9
```


If you make a mistake while typing, you can use the backspace key (CONTROL-H on most keyboards), or you can delete the entire line using the "line delete" key (usually CONTROL-X). There are several other control keys that have functions as listed below. Note: it is possible to redefine which keys correspond to these functions: see the TMODE command description.

Terminal Control Keys

CONTROL A - Repeat previous input line

CONTROL C - Program Interrupt
Many programs use this key to temporarily suspend program execution.

CONTROL D - redisplay present input line

CONTROL H - Backspace

CONTROL Q - Quit Program
This key can be used to abort execution of command programs (which returns you to the shell). This key is also used for other specific functions by BASIC, text editors, etc.

CONTROL W - display Wait
This control key will temporarily halt data output to your terminal so you can read the screen before the data scrolls off. Output is resumed when you hit any other key.

CONTROL X - line delete

ESCAPE (CONTROL [) - End-of-File
This key is used to send an end-of-file to programs that read input from the terminal in place of a disk or tape file.

Each shell command line begins with the name of a program, which can be followed by one or more optional "parameter" words or symbols. These are used to describe names of files or devices to be used by the program, or to activate optional functions or features.

The shell does not actually execute the command itself: what actually occurs is a particular program is called to perform the desired function. The program will be loaded from disk or tape if it is not already in memory.

AN INTRODUCTION TO MULTIPROGRAMMING

OS-9 is a "multiprogramming" (sometimes called "multitasking") operating system that has the ability to execute a number of independent programs ("processes") at the same time. Each process is independent and has its own memory space. OS-9 handles all communications between the process and the outside world. It also manages and allocates the three primary shared resources: memory, CPU time, and input/output devices.

The CPU is interrupted by a "real time clock" many times every second, which is a basic time interval called a "tick". At any occurance of a tick, OS-9 can suspend execution of one program and begin execution of another. The starting and stopping of programs is done in such a way that they are not affected and run as though they are not being interrupted.

Each "runnable" program is assigned one or more intervals of time per second to execute. Each interval is called a "time slice" and its duration depends on the process' priority, and also the number, states and priorites of other processes.

This technique is called "time-slicing" because each second of CPU time is sliced up to be shared among several processes. This happens so rapidly that to a human observer all processes appear to execute continuously, unless the computer becomes "overloaded" with processing. If this happens, a noticeable delay in response to terminal input may occur, or "batch" programs may take much longer to run than they ordinarily do.

CPU time is a limited resource that must be allocated wisely to maximize the computer's throughput. It is characteristic of many programs to spend a lot of unproductive time waiting for outside world events. A classic example is a program waiting for a line of input to be typed on a keyboard and spending 99.999% of the time waiting for a (relatively) slow human to hit keys. In order to avoid wasting time, processes are classifed into one of three categories:

ACTIVE PROCESSES are those which have useful work to do and are included in the timeslicing.

WAITING PROCESSES are suspended pending occurance of some event, usually completion of an input or output function, or a message from another process.

SLEEPING PROCESSES are suspended by self-request for a specified time interval.

Sleeping and waiting processes are not given CPU time until they are activated by some signal or event.

CREATION OF NEW PROCESSES

New processes are created when an existing process executes an OS-9 service request called "fork". One of the parameters passed in "fork" is the name of a program module the new process is to initially execute. The program can already be present in memory, or OS-9 may load it from a mass storage file having the same name. The shell's main function is to read an input line and perform a "fork" on the command (program) name given. This leads some important points about the shell:

1. There is nothing special about the shell: it is just another program as far as OS-9 is concerned. It is reentrant and many separate "incarnations" of the shell can exist at any given time.

2. The programs invoked by the shell are usually NEW PROCESSES that are capable of independent, concurrent execution.

3. The shell usually suspends itself from execution until termination of the program (process) it invoked. Then it wakes up, outputs another prompt, gets another input line, and performs another "fork". This loop repeats until end-of-file is detected on its input device, or the shell process is killed by external means.

Any process can create as many new processes as it wants, limited only by the amount of unassigned memory available to load and run new processes.

When processes create new processes, which in turn create additional new processes, a hierarchical lineage becomes evident. In fact, this hierarchy is a tree structure that resembles a family tree. The "family" concept makes it easy to describe relationships between processes, and so it is used extensively in descriptions of OS-9's multiprogramming operations.

When a process creates a new process, the original process is called the "parent" process and the newer process is called the "child" process. If a parent process creates more than one child process, the children are called "siblings" with respect to each other. Of course, the children can create their own children and become parents themselves, ad infinitum. Terminology such as "grandparents" and "great-grandparents" may be appropriate but are seldom used because we are usually more concerned with the "offspring" than the "ancestors".

PROCESS INITIALIZATION AND INHERITANCE

New processes are created when a process requests OS-9 to do so. The OS-9 "fork" service request is given the name of a program module the new process is to initially execute, and some optional parameters. OS-9 then takes over and starts the new process creation sequence. If any part of the sequence cannot be performed the fork is aborted and the prospective parent is notified. The most frequent reason for failure is when required rsources such as memory or I/O devices are not available at that instant.

OS-9's first step is to attempt to locate the requested program module by searching the "module directory", which has the name and type of every module already present in memory. The 6809 instruction set supports a type of program called "reentrant code" which means the exact same "copy" of a program can be shared by two or more different processes simultaneously without affecting each other, provided that each "incarnation" of the program has an independent memory area for its variables.

Most Microware OS-9 family software is reentrant and therefore makes most efficient use of memory. For example, the Macro Text Editor is about 4K in size. If you call for this program and some other user (process) has already caused it to be loaded, you will execute the same copy instead of loading another copy into an additional 4K of memory. OS-9 always keeps track of how many processes are using each program module and deletes the module (freeing its memory for other uses) when the last user is finished with it.

If the requested program module is not already in memory, the name is used as a pathlist (file name) and an attempt is made to load the program from mass storage.

Every program module has a "module header" that describes the program and its memory requirements. OS-9 uses this to determine how much memory for variable storage should be allocated to the process (this size can be optionally expanded using an optional parameter). The module header also includes other important descriptve information about the program, and is an essential part of OS-9 operation at the machine langauge level. A detailed decription of memory modules and module headers can be found in the "OS-9 System Programmer's Manual".

When a new process is created OS-9 assigns it a unique process ID code which is a number in the range of 1 to 255. It is also allocated the amount of memory space needed for its exclusive use. The new process also "inherits" some things from its parent:

1. A USER NUMBER which is used for file security and to identify processes belonging to a specific user. This is not the same as the "process ID", which identifies a specific process.

2. STANDARD INPUT AND OUTPUT PATHS: the files/devices the process uses for routine input and output, which typically corrspond to the keyboard and display.

3. OTHER FILES which the parent may have opened, and also the parent's current working directories (more on this later). Both the parent and the child can simultaneously access the same files.

4. PROCESS PRIORITY which determines what proportion of CPU time the process receives with respect to others.

You can use the "procs" command to see all the processes that exist under your user number and their status. It also shows the internal process ID number assignd by OS-9 to identify it. You can abort any of your processes using the "kill" command.

INPUT/OUTPUT: PATHS, FILES AND DEVICES

OS-9 has a unified input/output system which means data transfers to ALL I/O devices are done exactly the same way. At first glance it may seem that the previous statement may violate several laws of nature. After all, line printers and disk drives have vastly different operational characteristics. These differences can be overcome by defining a standardized LOGICAL STRUCTURE for all devices, and by making all I/O devices conform to this structure, using software routines to eliminate hardware-dependencies wherever possible.

This logical input/output structure is based on somthing called an "I/O PATH". It is defined as a description of a routing of data between a program (process) and a file or I/O device. The "program end" of the path is uniform all the time: but as data travels along the path it is routed and processed by OS-9 to conform to the hardware requirements of the specific I/O device. Data transfered through a path is considered to be a stream of binary bytes that have no specific type or context: what the data represents depends entirely on how it is used by programs.

The main advantages of this method are:

1. You can write programs that will operate correctly regardless of the particular I/O devices to be used when the program is executed.

2. Programs are highly portable from one computer to another, even when the computers have different kinds of I/O devices.

3. I/O can be redirected to alternate files or devices at run-time.

4. The same system calls used for physical input/output functions can be used for inter-process communications.

USING INPUT/OUTPUT PATHS

The OS-9 service requests "open" or "create" are used to establish new paths. The operating system is given a description of the "routing" of the path, which is called a "pathlist". The pathlist is a list of one or more names separated by slash "/" characters.

Names have up to 29 characters and may contain uppercase letters, lowercase letters, digits and underscore "_" characters. The first character of a name must be a letter.

ALL I/O DEVICES HAVE UNIQUE NAMES. The specific names used on a particular system are somewhat arbitrary because can be user defined, and depend on the type and number of I/O devices. For the purposes of the following examples, we will use the following fairly customary names:

"TERM" - the console terminal
"T1", "T2", etc. - other terminals
"P" - the printer
"DØ" - disk drive unit zero
"D1" - disk drive unit one

All devices have two basic attributes:

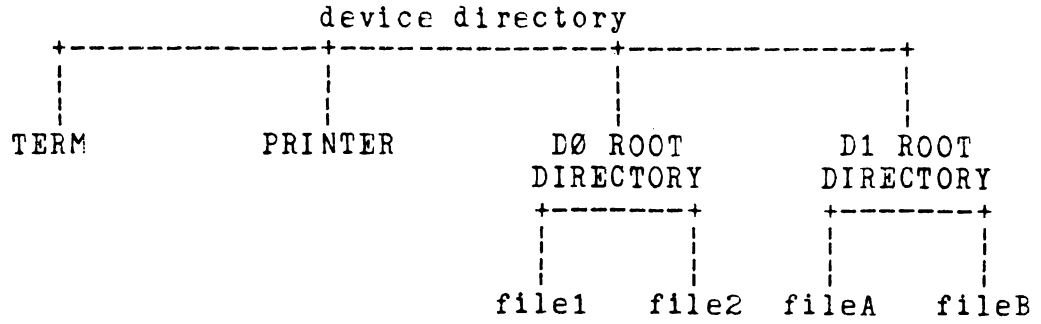
sharable or non-sharable
multifile or non-multifile

A device is considered sharable if more than one process can access it at the same time. Disks and terminals are usually sharable. Printers and tape systems are not usually sharable. OS-9 will not let a process establish an I/O path to a non-sharable device already in use by another process.

Multifile devices are mass storage devices such as disk systems that store data organized into individual entities called "files". Each file has a name that is kept in a directory. Every multifile device has a master directory (called the "root directory") that includes the names of the files and sub-directories stored on the device.

These classifications are not mutually exclusive. For example a tape system may be multifile but not sharable, a disk is sharable and multifile, a printer is neither sharable or multifile.

When OS-9 is asked to create an I/O path, it uses the names in the pathlist to search various directories to obtain the necessary routing information. These directories are organized as a tree-structured hierarchy. The highest-level directory is called the "device directory", which contains names and linkages to all the I/O devices on a given system. If any of the devices are of a multifile type they each have a master directory, which is the next-highest level. Here's what the tree of our example system looks like:



The names in this example device directory are "TERM", "PRINTER", "DØ" and "D1". DØ's root directory has two file names: "file1" and "file2". D1's root directory also has two file names: "fileA" and "fileB".

Let's use the copy command to illustrate construction and use of pathlists. The COPY command has the following form:

```
COPY  pathlist1  pathlist2
```

This command works by opening an input path using pathlist1, then opens an output path using pathlist2. Data is then read from the first path and written out on the second path. Suppose we want to get a print-out of the contents of "fileA" on device "D1" by copying it to "PRINTER". The actual command would look like this:

```
OS9: COPY  /D1/fileA  /PRINTER
```

Notice that the first pathlist uses two names: "D1" is a multifile device so another name must be given to specify a particular file. "PRINTER" is not multifile so one name is sufficient. Here's an example of a command that would duplicate "fileB" on device "D1" on a new file called "fileBcopy" on device "DØ":

```
OS9: COPY /D1/fileB  /DØ/fileBcopy
```


Here are some important rules that apply to usage of pathlists:

1. Pathlists cannot contain imbedded spaces or anything other than legal names or delimiters ("/" characters).

2. Pathlists that begin with a slash start directory searching with the root directory, otherwise the search starts in the "current working directory" - this is described later on.

3. The number of names in a pathlist must exactly correspond to the number of directories to be searched.

4. The same file name cannot appear more than once in any directory. Files with identical names can be used as long as the names are in different directories.

5. The "." or ".." symbols can be used at the beginning of a pathlist to refer to the current working directory and the parent directory of the current working directory, respectively.

MASS STORAGE FILES

Mass storage files are kept on multifile devices such as disks or some kinds of tape systems. The same file format is used regardless of the type or size of disk system used.

A file is an independent ordered sequence of data bytes stored on a multifile device. OS-9 is not concerned with what the bytes represent: they are treated as pure data. The data could be a machine language program, characters of text, or anything you want. Data is written to and read from files exactly as given. The file can be any size from zero up to the maximum capacity of the storage device. The theoretical limit is over four billion bytes. Files can be expanded or shortened as desired.

Creating New Files

An OS-9 service request called "create" is used to make new files. A pathlist is given with this request which specifies the directory in which the file name is to be placed, for example:

```
/D0/my_new_file
```

The file is also given some attributes which are described further on. The OS-9 "create" service request returns a unique path number that is used to refer to the file as long as it is open.

Accessing Existing Files

The OS-9 service request called "open" can be used to establish a path to an existing file or device according to a specified pathlist. It also returns a unique path number used in subsequent references to the file/device.

Reading and Writing Files

When a file is created or opened a "file pointer" is established for it. Bytes within the file are addressed like memory, and the file pointer holds the "address" of the next byte in he file to be written to or read from. The OS-9 "read" and "write" service requests always update the pointer as data transfers are performed.

Files can be read from and written to sequentially and/or randomly in any combination of ways. As mentioned above, the file pointer addresses the next data to be transferred, and the pointer is automatically updated in the process. Therefore, sucessive reads or writes will perform sequential data transfers.

A service request called "seek" can be used to reposition the file pointer to any byte address in the file. This is used when random access of the data is desired.

To expand a file, you can simply write past the previous end of the file. Reading up to the last byte of a file will cause the final "read" request to return an end-of-file status.

Compatibility With Non-Multifile Devices

The same OS-9 service requests that operate on mass storage files work on other kinds of devices as well. As mentioned above, the "open" and "create" requests are also used to establish paths to all other devices, the only difference being the routing given in the pathlist.

A natural question at this point is "how can you 'create' a printer, or 'seek' a terminal?". Obviously, the answer is that you can't, but these undefined operattions can be SIMULATED or IGNORED: and things usually work out OK. For example, when the printer driver routine receives a "create" request it does an equivalent operation: "open". When you ask a terminal driver to "seek" it simply ignores the request and does not return an error.

> IN THE VAST MAJORITY OF CASES, PROGRAMS INTENDED FOR USE WITH MASS STORAGE FILES WILL WORK CORRECTLY WITH NON-MULTIFILE DEVICES. THEREFORE, DEVICES AND FILES CAN BE USED INTERCHANGABLY.

This unusual consistancy of the OS-9 unified I/O system allows you to design programs that are highly portable and mostly hardware independent.

THE FILE SECURITY SYSTEM

As mentioned in the section describing "File Creation", files are given certain attributes when they are created. These attributes are:

Ownership
Date and Time Created
Date and Time of Last Access
Access Permissions

The ownership is established by storing the user number of the creating process. The dates and times of creation and last access are fairly self-explanatory. The main security function is based on the "access permissions".

There are six types of permissions that define how and by whom the file can be accessed. Each of the six can be turned "off" or "on" independently, BUT ONLY BY THE FILE'S OWNER. Here is a list of the access permissions:

WRITE PERMIT TO OWNER: If on, the owner may write to the file or delete it. This permission can be used to protect important files from accidental deletion or modification.

READ PERMIT TO OWNER: If on, the owner is allowed to read from the file. This can be used to prevent "binary" files from being used as "text" files.

EXECUTE PERMIT TO OWNER: If on, the owner can perform load, chain, and fork system calls using the file. This can be used to prevent "non-executable" data such as text to be accidentially used as program data, which has very nasty consequences.

The following "public permissions" work the same way as the "owner permissions" above but are applied to processes having DIFFERENT user numbers than the file's owner.

WRITE PERMIT TO PUBLIC

READ PERMIT TO PUBLIC

EXECUTE PERMIT TO PUBLIC

For example, if a particular file had all permissions on except "write permit to public" and "read permit to public", the owner would have unrestricted access to the file, but other users could execute it, but not read, copy, delete, or alter it.

HIERARCHICAL DIRECTORIES

We have already seen that there is a tree-structured ordering of at least two levels of directories: the system device directory is the highest level, and the multifile mass storage device "root directories" are the next level down.

It is possible to create a virtually unlimited number of levels of "subdirectories" on a mass storage device using the "makdir" system service request. There is also a shell command of the same name that does the same thing.

All multifile directories are actually slightly special files. They can be "opened" and read using the same service requests used for other files and devices, which can be very handy.

To demonstrate how directories work, we'll use the same example "system" and diagram shown on page 13. First, let's introduce the shell's DIR command which is used to examine the file names contained in a directory. If we enter the command:

```
OS9: DIR /DØ
```

we will get a list of the files on device DØ's root directory, which looks like this:

```
FILE NAMES IN DIRECTORY: /DØ

file1      file2
```

To make a new directory in this directory, we use the command:

```
OS9: MAKDIR /DØ/newdir
```

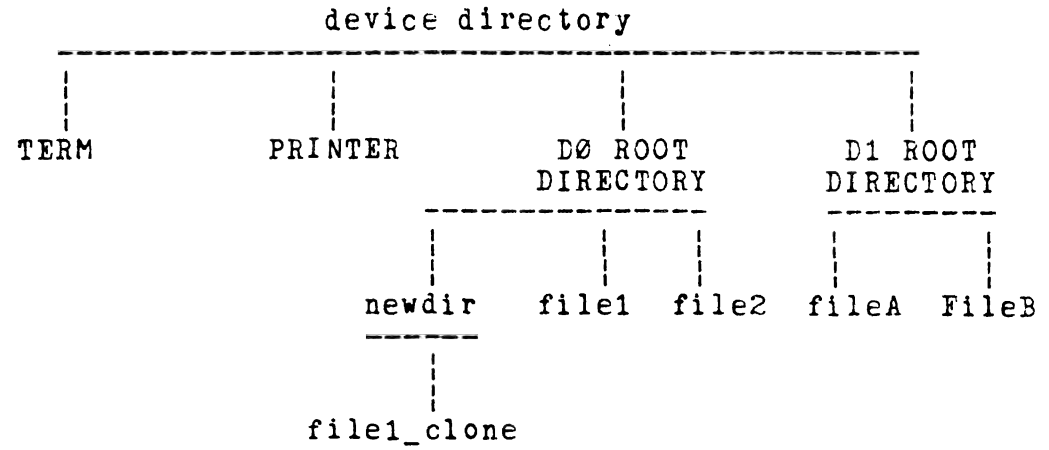
The directory file "newdir" is now a file listed in DØ's root directory:

```
OS9: DIR /DØ

FILE NAMES IN DIRECTORY: /DØ

file1      file2     newdir
```

So far, so good. Now lets create a new file and put in the new directory, using the COPY command to duplicate "file1":

```
OS9: COPY /DØ/file1 /DØ/newdir/file1_clone
```

Observe that the second pathlist now has three names: the name of the root directory ("DØ"), the name of the next lower directory ("newdir"), then the actual file name ("file1_clone"). Here's what the directories look like now:



We can use the DIR command to look at the files on our new directory:

```
OS9: DIR /DØ/newdir

FILE NAMES IN DIRECTORY: /DØ/newdir

file1_clone
```

It is possible to use MAKDIR to create a new directory in "newdir" which would add yet another level of directories to the system:

```
OS9: MAKDIR /DØ/newdir/another_directory
```

To copy a file to this directory we can do

```
OS9: COPY /D1/fileB /DØ/newdir/another_directory/fileX
```

It should be apparent by now that the lower the level of the file's directory, the more names are required in it's descriptive pathlist.

You may be thinking by now that a hierarchical file system is mostly good for improving your typing skills and otherwise complicating matters. This is really not so. The next section introduces powerful ways to change how directories are searched, which eliminates many long pathlists. This system is very important in a multiuser environment because it allows each user to privately organize his or her files as desired, without affecting other users.

Another advantage of the hierarchical directory system is that you can have files with identical names on the same device, providing that the names are in different directories. For example, you can have a set of test files to check out a program using the same file names as the program's actual working files, but there is no conflict because the files are in different directories.

Here are some characteristics of directories:

1. Directories have the same ownership and security attributes as regular files.

2. Each file is listed in exactly one directory.

3. Files can only be added to directories when they are created.

4. All files and directories from a device's root directory downward must reside on the save device.

5. You should never delete a directory file that contains file names. If you do so, the files (and their disk space) will be lost.

A very important point to remember about directory files is that they mostly behave like "normal" files. This means programs can easily read directories using the same means used to access other files.

USING AND CHANGING WORKING DIRECTORIES

Up to this point, we have used examples of pathlists that start searching at the system device directory (highest) level. In fact, we have forced this to occur because every pathlist used started with a "/" character.

Each process has two "working directories" associated with it. The simplest definition of a working directory is the directory where pathlist searching begins when a leading "/" is omitted from a pathlist. Any directory that the process is permitted to access can be made a working directory.

The two working directories are referred to as the "current execution directory" and the "current data directory". OS-9 uses the execution directory for service requests that load program modules into memory, such as load, fork and chain. This means the command files called by the shell come from this directory. The data directory is used for all other file references. The same directory can be used for both working directories.

If your system is multiuser, the log-on program will initially set your working directories to those specified in the system log-on data file.

Let's look at an example of a shell command statement using the default working directory notation, and its equivalent expansion:

```
COPY file1 file2
```

This is equivalent to:

```
OS9: /D0/SYS/CMDS/COPY /D0/JONES/file1 /D0/JONES/file2
```

Not only does the use of default directories save typing, but files are accessed faster because fewer (usually one) directories need be searched.

The shell commands "CHD" and "CHX" can be used to independently change the current working data and execution directories, respectively. These command names must be followed by a pathlist that describes the new directory file. You must have permission to access the directory according to normal file security rules. Here are some examples:

```
OS9: CHD /D1/new_data_dir

OS9: CHX /D0/sys_dir/new_exec_dir
```

Because working directories are inherited from a process' parent, the actual names may not be known, so special "name substitutes" are available. They are:

. always refers to the present working directory

.. always refers to the directory that contains the name of the present working directory (i.e., the next highest level directory)

They can be used in place of pathlists and/or names in pathlists. Here are some examples:

```
OS9: DIR .        lists file names in the working directory

OS9: DIR ..       lists names in the working directory's
                  parent directory.

OS9: DEL ../temp  deletes the file "temp" from the
                  working directory's directory.
```

The substitute names refer to either the execution or data directories, depending on the context in which they are used. For example, if ".." is used in a pathlist of a file which will be loaded and/or executed, it will represent the parent directory of the execution directory. Likewise, if "." is used in a pathlist describing a program's input file, it will represent the current data directory.

THE OS-9 SHELL

The shell is a command interpreter that reads input lines and processes them as requests to run other programs. The processing can also involve interpreting optional parameters that control or modify execution of programs. When the shell is first entered it prints:

```
Shell

OS9:
```

The "OS9:" is a prompt that indicates that the shell is expecting input. A command line can be typed which has the elementary form:

```
<program name> <parameters>
```

The "name" is that of the program to be executed. The program can be a module already in memory, or a complete pathlist describing a file that contains the desired program. The optional "parameters" are zero, one, or more sequences of text which are passed to the program. Here's an example of a simple command line:

```
OS9:  copy file1 file2
```

This calls the program "copy", which is passed two parameters, "file1" and "file2", which happen to be file names because this is what the "copy" program expects. Some programs expect parameters which are not file names; others expect (or optionally expect) parameters other than file names; others don't use any parameters at all.

Programs are executed by the shell using the "fork" system call, so they become separate, "child" processes of the shell. In the simple example above, the shell suspends itself until the "copy" process performs its function and terminates. The shell is then reactivated and prints a prompt for the next command line. The shell execution loop repeats until an end-of-file is encountered on its input file/device.

In addition to the basic input/execute function, the shell has several other important optional facilities that are used to modify program execution, run multiple programs simultaneously ("concurrent execution"), and execute procedure files. Each of these topics is explained in the following sections. There is a virtually unlimited number of ways these capabilities can be used, and is impossible to give more than a representative set of examples in this guide. You are therefore encouraged to study the basic rules, use your imagination, and explore the possibilities on your own.

Execution Modifiers: Alternate Memory and Redirected I/O

When command programs are invoked by the shell, they are allocated the minimum amount of working RAM memory specified in the program's module header. Sometimes it is desirable to increase this memory size, so the shell has a an optional memory size modifier that can be used for this purpose. For example, the copy program's module header indicates a requirement of 512 bytes of memory. To speed up the copy, we can assign it more memory. Memory can be assigned in 256-byte pages using the modifier "#n" where n is the number of pages, or in 1024 byte increments using the modifier "#nK". The two examples below behave identically:

```
OS9: copy #8 file1 file2

OS9: copy #2K file1 file2
```

In general, modifiers can be used before OR after the program's parameter list, if any.

The second kind of modifier is used to change the program's "standard I/O paths". There are three I/O paths for standard input, standard output, and standard error output. The name "standard error" output can be misleading because this path is also used for routine status messages, listings, etc. Correctly-written OS-9 programs use these paths for routine I/O. Because the programs do not use specific file or device names, it is fairly simple to "redirect" the I/O to any file or device without altering the program itself.

Recall that when processes are created by the "fork" system call, they inherit the parent process' standard I/O paths. Therefore, when the shell create new processes, they usually inherit its standard I/O paths. The redirected I/O modifiers cause new processses to be created with specific files/devices for standard I/O path(s) instead of inheriting the shell's. When you log-on the shell's standard input is the terminal keyboard; the standard output and error output is the terminal's display.

There are three modifers, "<", ">", and ">>" that "redirect" the standard input, output, and standard error output, repectively. They must be immediately followed by a pathlist describing the file or device the I/O is to be redirected to or from. For example, the standard output of "list" can be redirected to write to the system printer instead of the terminal:

```
OS9: list correspondence >/p
```

Files referenced in I/O redirection modifiers are automatically opened or created, and closed as appropriate. Here's another example:

```
OS9: list song >/d1/tune
```

The list command's output is redirected to a new file called "tune", which is functionally equivalent to the command:

```
OS9: copy song /d1/tune
```

Redirection modifiers can be used before or after the programs's parameters, but each modifier can only be used once per program. For example, the assembler uses the standard input, output, and error paths for its source file, object file, and listing file.

```
OS9: asm <pgm_src >pgm_obj >>/p
```

Command Separators: Sequential and Concurrent Execution

A single shell input line can request execution of more than one program. The program names, parameters, and modifiers must be divided by "separator" characters. The basic separator is the ";" which implies sequntial execution, meaning the preceding program is executed, and the shell waits for it to complete its function and terminate before the following program is executed. Carriage return characters that terminate shell input lines also operate as sequential execution separators with respect to the next input line. Here are some examples:

```
OS9: copy oldfile newfile; del oldfile; list newfile
```

```
OS9: asm #8k <src >obj >>temp; list temp >/p; del temp
```

The second kind of separator is the "&" which implies concurrent execution, meaning that the program is run but the shell does not wait for it to complete before starting the next one. This is the essential mechanism that performs multiprogramming on your command. The number of programs that can run simultaneously is not fixed: it depends upon the amount of free memory in the system versus the memory requirements of the specific programs. Here is an example:

```
OS9: dir >/printer& list file1& copy file2 file3; del file2
```

Because they were followed by "&" seperators, the "dir", "list", and "copy" programs will run concurrently, but the "del" program will not run until the "copy" program has terminated.

When the shell creates a new concurrent process, it prints an ampersand followed by the "process ID number" assigned to it. You can also get a list and status summary of all processes you have created by using the "procs" command.

The third kind of separator is the "!" character which is used to contruct "pipelines". Pipelines consist of two or more programs whose standard input and/or output paths connect to each other rather than to

files or I/O devices. These special kinds of paths are called "pipes" and are the primary means by which data is transfered from process to process. Pipes behave in exactly the same way as conventional files except I/O transfers between processes are automatically synchronized to the processes rather than to physical devices. Pipe seperators must always be followed by a program name. Here is an example pipeline:

```
OS9: update <master_file ! sort ! write_report >/p
```

In the example above, the program "update" has its input redirected from a path called "master_file". Its standard output is becomes the standard input for the program "sort". Its output, in turn, becomes the standard input for the program "write_report", which has its standard output redirected to the printer.

All the programs execute concurrently, and the I/O is automatically synchronized so the output of one program never "gets ahead" of the input request of the next program in the pipeline. This implies that data cannot flow through a pipeline any faster than the slowest program can process it. Some of the most useful applications of pipelines are jobs like character set conversion, print file formatiing, data compressions/decompression, data monitoring, etc. These types of "in-line" programs are often called "filters".

Note that your system may not have a pipe file manager: it is standard with OS-9 Level Two systems, but optional with OS-9 Level One systems.

Command Grouping

Sections of shell input lines can be enclosed in parentheses which permits modifiers and separators to be applied to an entire set of programs. The shell processes them by calling itself recursively (as a new process) to execute the enclosed program list. For example:

```
OS9: (dir /dØ; dir /d1) >/p&

OS9: (list f1; list f2)& (del obj; asm #8k <prog2 >obj) >/p&
```

Shell Pseudo-Commands and Options

When processing input lines, the shell looks for several special names of "pseudo-commands": commands that are built-in to the shell and are executed without creating a new process. They generally affect how the shell operates. They can be used at the beginning of a line, or following any program separator (";", "&", or "!"). Two or more adjacent pseudo-commands can be separated by spaces or commas.

The pseudo-commands and their functions are:

chd <pathlist>	change the working data directory to the file specified by the pathlist.
chx <pathlist>	change the execution directory to the file specified by the pathlist
ex name	directly execute the module named. This transforms the shell process so it ceases to exist.
w	wait for any process to terminate.
* text	comment: "text" is not processed.
kill <proc ID>	sends abort signal to process
setpr <proc ID> <priority>	changes process' priority
x	causes shell to abort on any error
-x	causes shell not to abort on error (default)
p	turns shell prompt and messages on (default)
-p	inhibits shell prompt and messages
t	makes shell copy all input lines to output
-t	does not copy input lines to output (default)

The change directory commands switch the shell's working directory and, by inheritance, any subsequently created child process. The "ex" command is used where the shell is needed to initiate execution of a program without the overhead of a suspended "shell" process. The name used is processed according to standard shell operation, and modifiers can be used.

Execution of Procedure Files

The shell is a reentrant program that can be simultaneously executed by more than one process at a time. As is the case with most other OS-9 programs, it uses standard I/O paths for routine input and output. Specifically, its requests command lines from the standard input path and writes its prompts and other data to the standard error path.

The shell can create a new process that also executes the shell program. By redirecting the standard input path to a mass storage file, the shell can accept and execute command lines from the file instead of a terminal keyboard. The text file to be processed is called a "procedure file". Its contents are one or more command lines that are identical to command lines that are manually entered from the keyboard. This technique is sometimes called "job stream processing".

Execution of procedure files has a number of valuable applications. It can eliminate manual entry of commonly-used sequences of commands. It can allow the computer to execute a lengthy series of programs "in the background" while the computer is unattended or while the user is running other programs "in the forground".

In addition to redirecting the shell's standard input to a procedure file, the standard output and standard error output can be redirected to another file which can record output for later review or printing. This can also eliminate the sometimes-annoying output of shell messages to your terminal at random times.

Here are two simple ways to use the shell to create another shell:

```
OS9: shell <procfile

OS9: () <procfile
```

Both do exactly the same thing: execute the commands of the file "procfile". To run the procedure file in a "background" mode you simply add the ampersand operator:

```
OS9: () <procfile&
```

OS-9 does not have any constraints on the number of job streams that can be simultaneously executed as long as there is memory available. Also, the procedure files can themselves cause sequential or concurrent execution of additional procedure files. Here's a more complex example of initiating two processing streams with redirection of each shell's standard output to files:

```
OS9: (t -e) <proc1 >>stat1& (t -e)<proc2 >>stat2&
```

Note that the pseudo-commands "-e" (inhibit prompts) and "t" (copy input lines to output) were used above. They make the output file contain a record of all lines executed, but without useless "OS9" prompts intermixed. The "x" pseudo-command can be used if you want the processing to stop if an error occurs. Note that the pseudo-commands only affect the shell that executes them, and not any others that may exist.

Using the Shell For Timesharing

OS-9 systems used for timesharing usually have a procedure file that brings the system up by means of a one simple command. A procedure file is created that creates one process for each terminal. The procedure file first starts the system clock, then initiates concurrent execution of a number of processes that have their I/O redirected to each timesharing terminal. Each process executes "tsmon", which is a special secure program that monitors the terminal for activity, then initiates the "login" program. Here's a sample procedure file for a 3-terminal timesharing system having terminals names "T1", "T2", and "T3".

```
* system startup procedure file
setime
tsmon t1
tsmon t2
tsmon t3
echo ** timesharing system now on-line **
```

SYSTEM COMMAND DESCRIPTIONS

This section of the User's Guide contains descriptions for each of the standard command programs. These programs are usually called using the shell, but can be called from many other OS-9 family programs such as BASIC09, Interactive Debugger, Macro Text Editor, etc. Unless otherwise noted, these programs are designed to run as individual processes.

Formal Syntax Notation

Each command description includes a syntax definition which describes how the command sentence can be constructed. These are symbolic descriptions that use the following notation:

[ ] = brackets indicate that the enclosed item(s) are optional.

{ } = braces indicate that the enclosed item(s) can be optionally repeated

<path> = represents any legal pathlist

<devname> = represents any legal device name

<modname> = represents any legal memory module name

<procID> = represents a process number

<opts> = one or more options defined in the command description

<arglist> = a list of arguments (parameters)

<text> = a character string termminated by end-of-line

attr Change File Attributes

Syntax: attr <path> { <opts> }

Allows the security permssions of a file to be changed. Each permission can be turned on or off. The name of the attribute is given to turn it on, or turn it off if preceded by a minus sign. Permissions not named are not affected. You are not allowed to change the permisssions of a file you do not own.

The file permission abbreviations are:

```
 d = directory file
 s = sharable file
 r = read permit to owner
 w = write permit to owner
 e = execute permit to owner
pr = read permit to public
pw = write permit to public
pe = execute permit to public
```

The "attr" command can be used to change a directory file to a non-directory file, in fact this is the only way a directory can be deleted. Use extreme caution: deleting a directory that has file name(s) in it will cause the loss of the file(s) and the loss of the their storage space until the media is reformatted. You cannot change a non-directory file to a directory file with this command.

Examples:

```
attr myfile -pr -pw

attr myfile r w e pr rw pe

attr /d1/temp r -pr w -pw

attr bonnies_dir -d
```

backup make backup copy of media

Syntax: backup [e] [s] [<devnam> [<devname>]]

This command is used to physically copy all data from one device to another. A physical copy is performed sector by sector without regard to file structures. In almost all cases the devices specified must be of the same type.

If both device name are omitted the names "/DØ" and "/d1" are assumed. If the second device name is omitted, a single unit backup will be performed.

The options are:

```
   e = exit if any read error occurs
   s = print single drive prompt messaged
 #nK = more memory makes backup run faster
```

Examples:

```
backup /tape1 /tape2

backup /DØ /D1
```


build build a new text file

Syntax: build <path>

Creates a new file using the pathlist given and copies text lines from the keyboard (standard input path) to the file. The command terminates when a line is entered that is just a carriage return.

Example:

```
build small_file
build >/p          (copies keyboard to printer)
```

chd — Change Working Data Directory
chx — Change Working Execution Director

Syntax: chd <pathlist>
chx <pathlist>

These are shell pseudo-commands that change the user's current working directories. See the "shell" description for more information.

Examples:

```
chd /d1/master_accounts.
chx ..
chx binary_files/test_programs
chx /D0, chd /D1
```

copy .copy data from a path to another

Syntax: copy <path> <path>

Copies binary data from the file or device specified to the second file or device specified. The first file/or device must already exist, the second file is created if the device is multifile. Data is NOT modified in any way as it is copied.

Data is transfered using large block reads until end-of-file occurs on the input path. Because block transfers are used, normal output processing of data does not occur on character-oriented devices such as terminals, printers, etc., Therefore, the LIST command is preferred over COPY when data consisting of text is to be copied.

Using the shell's alternate memory size modifier to give copy a large memory space can often increase its speed.

Examples:

```
copy file1 file2
copy /D1/joe/news /printer&
```

date display system date

Syntax: date [t]

Prints the current system date. If the "t" option is given the current time is also displayed.

Example
```
date t

March 14, 1980  08:42:12
```

del delete a file

Syntax: del <path> { <path> }

The file(s) specified are deleted. The pathlist(s) must describe file(s) on multfile device(s) for which the user has write permission. Directory files cannot be deleted unless their type is changed first: see the "attr" command description.

Examples

```
del test_program old_test_program

del /D1/number_five
```

dir display names in a directory file

Syntax: dir [e] [<path>]

Displays the names of files listed in the directory file specified by the pathlist, or the current working data directory if the pathlist is omitted.

If the "e" option is included, each file's entire description is displayed: size, address, owner, permissions, date and time of last modification.

The following forms of the dir command are epecially useful:

DIR . displays the current directory

DIR .. displays the parent directory of the current working directory.

Examples:

```
dir /d1/jane

dir new_stuff

dir e test_programs
```


dump perform file dump

Syntax: dump [<path>]

Produces a formatted display of the actual data on the mass storage file specified. The data is displayed in both hexadecimal and ASCII character format. Data bytes that have non-displayable values are represented by periods in the character area. If a pathlist is omitted, dump uses standard input.

The addresses displayed on the dump are relative to the beginning of the file. Because memory modules are position-independent and stored on files exactly as they exist in memory, the addresses shown on the dump correspond to the relative load addresses of memory-module files.

```
OS9: dump shortfile
```

echo Echo Input Line to Output

Syntax: echo <text>

This command echoes its argument to the standard output path. It is mostly used to generate informational messages in shell procedure files.

Examples:

```
echo >/term ** warning ** disk about to be scratched!

echo >/p Listing of Transaction File; list trans >/p
```

ex Execute Program

Syntax: ex <module name> [<modifiers>] [<parameters>]

This a shell pseudo-command that changes the process from execution of the shell to execution of another program. It permits a transition from the shell to another program without creating another process. This command is often used when the shell is called from another program to execute a single, specific program, after which the shell is not needed.

The "ex" command should always be the last command on a shell input line. See the "shell" description for more information.

Examples:

```
ex BASIC09

dir /d1; ex tsmon
```

format — Format a New Diskette

Syntax: format <devname> [<option list>]

Used to physically initialize, verify, and establish an initial file structure on floppy disk media. This command must be used on new media before it can be used in an OS-9 system.

This command can format many types of disks. A physical decription can be given in the parameter list. If parameters are ommitted, the program will ask a series of questions. The parameters are:

```
S = single density (default)
D = double density
1 = single sided (default)
2 = double sided
5 = five inch media
8 = eight inch media
R = inhibit ready prompt
/name/ = disk name (32 character maximum)
```

The formatting process works as follows:

1. The physical format is written on the disk.

2. Each sector is read back and verified. If a sector fails the test it is "locked" out of the initial free space on the disk. As the verification is performed, track numbers are displayed on the standard output device.

3. The disk allocation map, root directory, identification block, and bootstrap file are written on the disk.

Examples:

```
OS9: format 5 d 2 /database/ r


OS9: format
DOUBLE DENSITY?  N
DOUBLE SIDED?  N
5" or 8": 8
READY: Y
(track numbers displayed here)
GOOD SECTOR COUNT = 04D0
DISK NAME: jim's files
FORMAT COMPLETE
```

free Display Free Space on Device

Syntax: free <devname>

Displays the number of 256-byte sectors of unused space available for new files or expanding existing files. The device name given must be that of a mass-storage multifile device.

Data sectors are allocated in groups called "clusters". The number of sectors per cluster depends on the storage capacity and physical characteristics of the specific device. This means that small amounts of free space may not be divisible into as many files. For example, if a given disk system uses 8 sectors per cluster, and a "free" command shows 32 sectors free, a maximum of four new files could be created, if each has only one cluster.

Example:

```
OS9: free

BACKUP DATA DISK created on: 12 July 1980
Capacity: 1,232 sectors (1-sector clusters)
1,020 free sectors, largest block 935 sectors
```

kill Abort a Process

Syntax: kill <procID>

Sends an "abort" signal to the process having the process ID number specified. The process must exist and have your user ID. The "procs" command can be used to obtain the process ID numbers of all of your processes.

Examples:

```
kill 5
kill 22
```

list List Contents of Text File

Syntax: list <path>

This command copies text lines from the input path specified to the standard output path. The program terminates upon reaching the end-of-file of the input path.

Examples:

```
list /D1/user5/document

list animals
```

login Log User Into System

Syntax: login

This program asks for a user name and password, which is checked against a validation file. If the information is correct, the user's system priority, user ID, and working directories are set up according to information stored in the file, and the shell is executed. If the user cannot supply a correct user name and password after three attempts, the process is aborted.

The validation file is called "password" and must be present in the current data directory. The file contains one or more variable-length text records, one for each user name. Each record has the following fields, which are delimited by commas:

1. User name (up to 32 characters, may include spaces)
   If omitted, anything will match.
2. Password (up to 32 characters, may include spaces)
   If omitted, no password will be requested
3. User ID number: must be uniquely assigned from Ø to 65535.
   This is the number used by the file security system.
4. Initial process priority: 1 - 255
5. Pathlist of initial execution directory.
6. Pathlist of initial data directory.
7. Name of initial program to execute (usually "shell")

Here's a sample validation file:

```
linda,hello computer,2,128,linda_ex_dir,linda_dat_dir,shell
ted,howdy,3,128,ted_ex_dir,ted_dat-dir,shell
jan,smith,10,200,order_entry_sys,order_files,basic09 #12K menu
george,wiz,1,255,.,.,shell t<startup; ex shell
```


load Load Modules Into Memory

Syntax: load <path>

The file specified is opened and one or more modules is read from it and loaded into memory. The names of the modules are added to the module directory. If a module is loaded that has the same name and type as a module already in memory, the module having the highest revision level is kept.

Example:

```
load new_program
```

makdir Make a New Directory File

Syntax: makdir <path>

Creates a new directory file according to the pathlist given. The pathlist must refer to a parent directory for which the user has write permission. The new directory is initialized and linked to is parent directory. It is given the access permissions: read to user, write to user, and read to public.

Examples:

```
makdir /d1/old_dir/new_dir     (creates new_dir)

makdir zap    (working directory is parent directory)
```

mdir display module directory

Syntax: mdir [e]

Displays the present module names in the system module directory, i.e., all modules currently resident in memory.

If the "e" option is given, a full listing of the address, size, type, revision level, and user count of each module is displayed. All numbers shown are in hexadecimal.

WARNING: not all modules listed by MDIR are executable as processes: always check the module type code before executing a module if you are not familiar with it.

mfree list free memory areas

Syntax: mfree

Displays a list of which areas of memory are available for assignment. The address and size (number of 256-byte pages) of each "hole" is listed. The addresses are shown in hexadecimal and the sizes in decimal.

Example:

```
  mfree

  Address  pages
 --------- -----
 1000-1BFF   12
 1600-B8FF  163
 C000-C0FF    1

 Total pages free = 176
```

procs Display Procedure and Status

Syntax: procs [e]

Generates a listing of processes presently in existance. If the "e" option is used all processes in the system are listed, otherwise only those having the caller's user ID.

The process ID number, status, priority, and primary program module is given for each process.

The status shown is a "snapshot" taken at the instant the command is executed: processes can switch states rapidly, sometimes several times per second. The priority is the value used by the OS-9 CPU scheduler to determine the length and frequency of the process' time slice. The primary module is the name of the program the process initially executed when it was invoked. All numbers displayed are decimal.

Example:

```
  ID Pty  State   Primary Module
 --- --- -------- --------------
   2  10  waiting Shell
   1  32 sleeping TSMon
   3  10  waiting Sysgo
   6  25   active Basic09
```

rename — Change file name

Syntax: rename <path> <new name>

Gives the mass storage file specified in the pathlist a new name. The user must have write permission for the file to change its name.

Examples:

```
rename blue purple

rename /D3/user9/test temp
```

save Copy Memory Module to File

Syntax: save <path> <modname> {<modname>}

Creates a new file and writes a copy of the memory module(s) specified on to the file. The module name(s) must exist in the module directory when saved. The new file is given access permissions for all modes except public write.

Examples:

```
save wordcount wcount
save math_pack add sub mul div
```

setime — activate and set system clock

Syntax: setime [y,m,d,h,m,s]

Inputs the date and time, then activates the real time clock. The numbers are one or two digit decimal using space, colon, semicolon or slash delimiters. If the time is not given in the command line setime will print a prompt requesting the data. The hours count up to 24, i.e., 15:20 is 3:20 PM.

IMPORTANT NOTE: This command must be executed before OS-9 can perform multitasking operations. If the system does not have a real time clock this command should still be used to set the date for the file system.

Examples:

```
setime 80,6,22,14,2,0

setime 81/02/12 8:43:20
```

setpr set process priority

Syntax: setpr <procID> <number>

This command is used to set a process' priority, which is a value used by the OS-9 CPU time scheduler to determine the duration and frequency of your time slices. The id number must be of a process that has your user ID.

The value is a decimal number in the range 1 to 255.

Example:

```
setpr 8 250
```

shell OS-9 Command Interpreter

Syntax: shell <arglist>

The shell is a program that reads data from its standard input path and interprets the data as a sequence of commands. The basic function of the shell is to initiate and control execution of other OS-9 programs.

The shell reads and interprets one text line at a time from the standard input path. After interpretation of each line it reads another until an end-of-file condition occurs, at which time it terminates itself. A special case is when the shell is initially entered: it interprets its argument list as its first input line.

The rest of this description is a technical specification of the shell syntax. Refer to the "Shell" section of this guide for a user-oriented discussion of how to use the shell.

Shell Input Line Formal Syntax:

```
<pgm line> := <pgm> {<pgm>}
<pgm> := [<params>] [ <name> [<modif>] [<pgm params>] [<modif>] ] {<sep>]
```

Program Specifications

```
<name>  := <module name>
        := <pathlist>
        := ( <pgm list> )
```

Parameters

```
<params>:= <param> { <delim> <param> }
<delim> := space or comma characters
<param> := ex <name> [<modif]    chain to program specified
        := chd <pathlist>        change working directory
        := kill <procID>         send abort signal to process
        := setpr<procID> <pty>   change process' priority
        := chx <pathlist>        change execution directory
        := w                     wait for any process to die
        := p                     turn "OS9: " prompting on
        := -p                    turn prompting off
        := t                     echo input lines to std output
        := -t                    don't echo input lines
        := -x                    don't abort on error
        := x                     abort on error
        := * <text>              comment line: not processed
```

shell - continued

Separators

```
<sep>    :=  ;     sequential execution separator
         :=  &     concurrent execution separator
         :=  !     pipeline separator
         :=  <cr>  end-of-line (sequential execution separator)
```

Modifiers

```
<modif> := <mod> { <delim> <mod> }
<mod>   := < <pathlist>     redirect standard input
         := > <pathlist>     redirect standard output
         := >> <pathlist>    redirect standard error output
         := # <integer>      set process memory size in pages
         := # <integer> K    set program memory size in 1K increments
```

tmode Change Terminal Operational Mode

Syntax: tmode [.<pathnum>] [<arglist>]

This command is used to display or change the operating parameters the user's terminal.

If no arguments are given, the present values for each parameter are displayed, otherwise, the parameter(s) given in the argument list are processed. Any number of parameters can be given, and are separated by spaces or commas.

A period and a number can be used to optionally specify the path number to be affected. If non is given it defaults to the standard input path.

upc	Upper case only. Lower case characters are automatically converted to upper case.
-upc	Upper case and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
-bsb	no erase on backspace: echoes single backspace only
bsl	Backspace over line: lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line: lines are "deleted" by printing a "new line" sequence (for hard-copy terminals).
echo	input characters "echoed" back to terminal (default)
-echo	no echo
lf	auto line feed on: line feeds automatically echoed to terminal on input and output carriage returns (default).
-lf	auto line feed off.
pause	screen pause on: output suspended upon full screen. See "pag" parameter for definition of screen size. Output can be resumed by typing any key.
-pause	screen pause mode off.
null=n	set null count: number of null ($00) characters transmitted after carriage returns for return delay. Number in decimal. default = 0.
pag=n	set video display page length to n (decimal) lines. Used for "pause" mode, see above.

tmode - continued

bsp=h set input backspace character. Numeric value of character in hexadecimal. Default = Ø8.

bse=h set output backspace character. Numeric value of character in hexadecimal. Default = Ø8.

del=h set input delete line character. Numeric value of character in hexadecimal. Default = 18.

bell=h set bell (alert) output character. Numeric value of character in hexadecimal. Default = 7

eor=h set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = ØD

eof=h set end-of-file input character. Numeric value of character in hexadecimal. Default = 1B.

type=h ACIA initialization value: sets parity, word size, etc. Value in hexadecimal. Default = 15.

Examples:

```
tmode -upc lf null=4 bse=1F pause

tmode pag=24 pause bsl -echo bsp=8 bsl=C
```

tsmon timesharing terminal monitor

Syntax: tsmon [<pathlist>]

Used to monitor idle terminals in timesharing applications. If a pathlist is given, it is opened in "update" mode as the standard I/O paths. It then waits for a carriage return character to be typed. It then performs a "fork" to the "login" command. If the login fails because the user could not supply a valid user name or password, login will return to tsmon.

Note: login and its password file must be present for tsmon to work correctly.

unlink Release Memory Module

Syntax: unlink <modname> { <modname>}

Tells OS-9 that the memory module(s) named are no longer needed by the user. The module(s) may or may not be destroyed and their memory reassigned, depending on if in use by other processes or user, whether resident in ROM or RAM, etc.

It is good practice to unload modules whenever possible to make most efficient use of available memory resources.

Warning: never unlink a module you did not load or link to.

Example:

```
unlink pgm1 pgm5 pgm99
```

OS-9 ERROR CODES

The error codes shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

C8	200	PATH TABLE FULL - The file cannot be opened because the system path table is currently full.
C9	201	ILLEGAL PATH NUMBER - Number too large or for non-existant path.
CA	202	INTERRUPT POLLING TABLE FULL
CB	203	ILLEGAL DEVICE - Can't find device descriptor, file manager or device driver.
CC	204	DEVICE TABLE FULL - Can't add another device.
CD	205	ILLEGAL MODULE HEADER - Module's sync code, check character or CRC is incorrect.
CE	206	MODULE DIRECTORY FULL - Can't add another module
CF	207	MEMORY FULL - Not enough contiguous RAM available to process request.
D0	208	ILLEGAL SERVICE REQUEST - System call had an illegal code number.
D1	209	MODULE BUSY - non-sharable module is use by another process.
D2	210	BOUNDARY ERROR - Memory allocation or deallocation request not on page boundary.
D3	211	END OF FILE - End of file encountered on read.
D4	212	NOT YOUR MEMORY - attempted to deallocate memory not previously assigned.
D5	213	NON-EXISTING SEGMENT - device has damaged file structure.
D6	214	NO PERMISSION - you don't have owner's permission to access the file as requested.
D7	215	BAD PATH NAME - syntax error in pathlist
D8	216	MISSING PATHLIST - expected pathlist missing or in error
D9	217	SEGMENT LIST FULL - file is too fragmented to be expanded further.
DA	218	FILE ALREADY EXISTS - file name already appears in current directory.
DB	219	ILLEGAL BLOCK ADDRESS - device's file structure has been damaged.
DC	220	ILLEGAL BLOCK SIZE - device's file structure has been damaged.
DD	221	MODULE NOT FOUND - request for link to module not found in directory.

OS-9 ERROR CODES - CONTINUED

DE	222	SECTOR OUT OF RANGE - device file structure damaged or incorrectly formatted.
DF	223	SUICIDE ATTEMPT - request to return memory where your stack is located.
E0	224	ILLEGAL PROCESS NUMBER - no such process exists.
E1	225	ILLEGAL SIGNAL CODE
E2	226	NO CHILDREN - can't wait because process has no children.
E3	227	ILLEGAL SWI CODE - must be 1 to 3.
E4	228	KEYBOARD ABORT - process aborted by signal code 2
E5	229	PROCESS TABLE FULL - can't fork now.
E6	230	ILLEGAL PARAMETER AREA - hi and low bounds passed in fork call are incorrect.
E7	231	BACKTRACK ERROR - you'll never see this one.
E8	234	SIGNAL ERROR - receiving process has previous unprocessed signal pending.
F0	240	UNIT ERROR - device unit does not exist.
F1	241	SECTOR ERROR - sector number is out of range.
F2	242	WRITE PROTECT - device is write protected.
F3	243	CRC ERROR - CRC error on read or write verify
F4	244	READ ERROR - hardware error during disk read operation
F5	245	WRITE ERROR - hardware error during disk write operation
F6	246	NOT READY - device has "not ready" status
F7	247	SEEK ERROR - physical seek to non-existant sector
F8	248	MEDIA FULL - insufficient free space on media
F9	249	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)

Command Program Syntax Summary

attr <path> {<opts>}	set file attributes
backup <devname> <devname>	backup media
build <path>	copy std. input to file
chd <path> *	change data directory
chx <path> *	change execution directory
copy <path> <path>	copy path to path
date [t]	display date (and opt. time)
del <path>	delete file
dir [<path>]	list file names in directory
dump <path>	binary dump of file contents
echo <text>	write text to std. output
ex <arglist> *	replace shell with program
format <devname> [<options>]	format media
free <devname>	give free space size
kill <procID> *	abort process
list <path>	list text file to std. output
load <path>	load module(s) into memory
login	log user into system
makdir <path>	create new directory
mdir [e]	list memory module directory
mfree	give free memory size(s)
p *	turn shell prompts on/off
procs	show processes and status
rename <path> <new name>	change file name
save <path> [<modname>] {<modname>}	save memory module(s) on file
setime [<y,m,d,h,m,s>]	activate and set system clock
setpr <procID> <number> *	change process priority
shell [<arglist>]	command interpreter
t *	copy shell input to output
tmode [<arglist>]	show/set terminal parameters
tsmon [<devnam>]	timesharing idle process
unlink <modname> { <modname>}	release memory modules
w *	wait for any process to die
x *	abort shell on error

*indicates shell pseudo-command

Optional OS-9 Family Software Command Syntax

asm [<options>]	interactive assembler
basic09 [<path>]	BASIC09
debug	interactive debugger
edit [<path> [<path>]]	macro text editor
mbasic	Microsoft basic
styl [<path> [<path>]]	Stylograph word processor